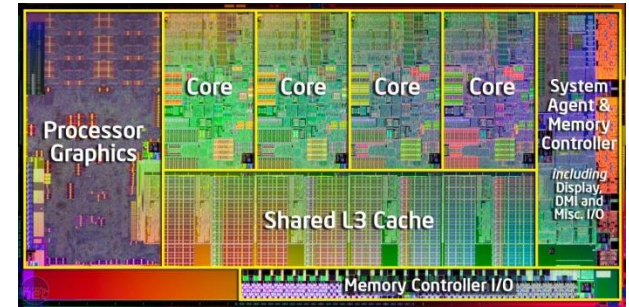




고려대학교
KOREA UNIVERSITY



COSE222, COMP212 Computer Architecture

Lecture 4. MIPS & MIPS Instructions #1

Arithmetic and Logical Instructions

Prof. Taeweon Suh

Computer Science & Engineering

Korea University

CISC vs RISC

- CISC (Complex Instruction Set Computer)

- One instruction does **many (complex)** jobs
 - Example: `movs` in x86
- Variable length instruction
- Examples: x86 (Intel, AMD), Motorola 68k



- RISC (Reduced Instruction Set Computer)

- Each instruction does a **small (unit)** job
 - Example: `lw`, `sw`, `add`, `slt` in MIPS
- Fixed-length instruction
- Load/Store Architecture
- Examples: MIPS, ARM

MIPS
TECHNOLOGIES

ARM[®]

MIPS

- Stanford University led by John Hennessy started work on MIPS in 1981
 - John is currently the president of Stanford Univ.
- MIPS has 32-bit and 64-bit versions
 - We study a 32-bit version
- MIPS is currently used in many embedded systems
 - Android supports MIPS hardware platforms
 - DVD, Digital TV, Set-Top Box
 - Nintendo 64, Sony Playstation and Playstation 2
 - Cisco routers
- Check out the MIPS web page for more information
 - www.mips.com





Android

Home screen displayed by Samsung Nexus S with Google, running Android 2.3 "Gingerbread"

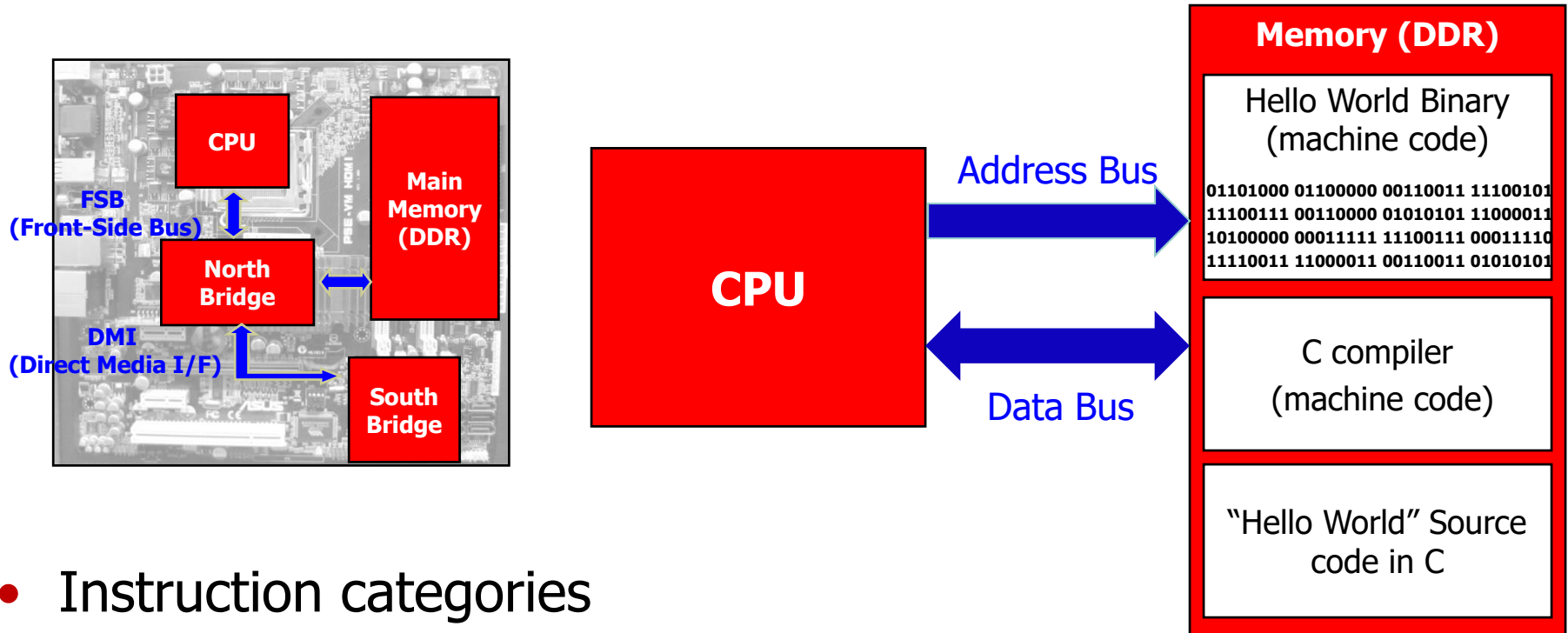
Company / developer	Google Inc, Open Handset Alliance
Programmed in	C (core), ^[1] Java (UI), C++
Working state	Current
Source model	Mixed (free and open source software and proprietary software) ^{[2][3]}
Initial release	21 October 2008
Latest stable release	Tablets: 3.2 (Honeycomb) ^[4] Phones: 2.3.6 (Gingerbread) / 2 September 2011; 11 days ago ^[4]
Package manager	APK
Supported platforms	ARM, MIPS, ^[6] x86 ^[8]
Kernel type	Linux kernel (monolithic)
Default user interface	Graphical
License	Apache License 2.0 before 3.0, closed source for 3.0, 3.1 and 3.2; Linux kernel patches under GNU GPL v2 ^[7]
Official website	android.com 

MIPS (RISC) Design Principles

- **Simplicity favors regularity**
 - Fixed size instructions
 - Small number of instruction formats
 - Opcode always occupies the first 6 bits in instructions
- **Smaller is faster**
 - Limited instruction set
 - Limited number of registers in register file
 - Limited number of addressing modes
- **Make the common case fast**
 - Arithmetic operands from the register file (load-store machine)
 - Allow instructions to contain immediate operands
- **Good design demands good compromises**
 - Three instruction formats

MIPS Instructions

- Let's discuss what kinds of instructions are essential to CPU



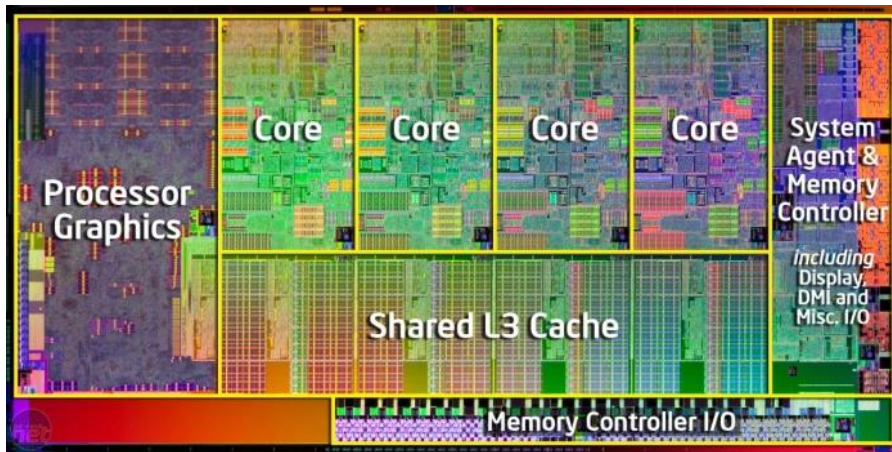
- Instruction categories
 - Data processing instructions (Arithmetic and Logical)
 - Memory access instructions (Load/Store)
 - Branch instructions

MIPS Essential Instructions

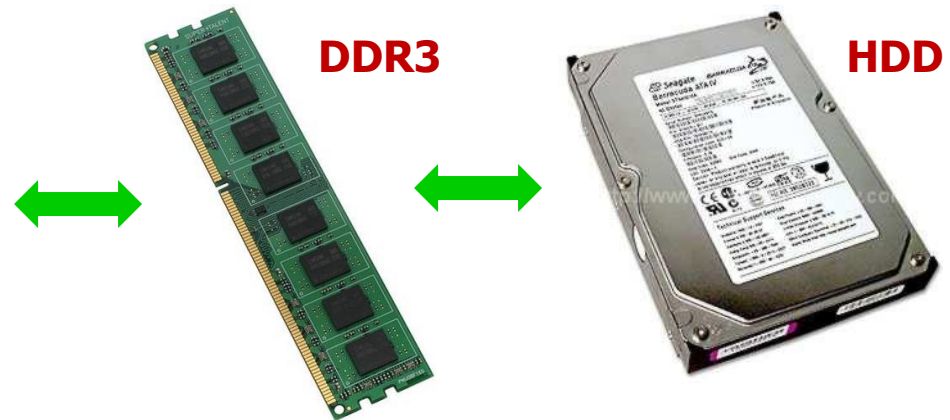
MIPS assembly language				
Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Three register operands
	subtract	sub \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Three register operands
	add immediate	addi \$s1,\$s2,20	$\$s1 = \$s2 + 20$	Used to add constants
Data transfer	load word	lw \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Word from memory to register
	store word	sw \$s1,20(\$s2)	$\text{Memory}[\$s2 + 20] = \$s1$	Word from register to memory
	load half	lh \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Halfword memory to register
	load half unsigned	lhu \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Halfword memory to register
	store half	sh \$s1,20(\$s2)	$\text{Memory}[\$s2 + 20] = \$s1$	Halfword register to memory
	load byte	lb \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Byte from memory to register
	load byte unsigned	lbu \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Byte from memory to register
	store byte	sb \$s1,20(\$s2)	$\text{Memory}[\$s2 + 20] = \$s1$	Byte from register to memory
	load linked word	ll \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Load word as 1st half of atomic swap
Logical	store condition. word	sc \$s1,20(\$s2)	$\text{Memory}[\$s2 + 20] = \$s1; \$s1 = 0 \text{ or } 1$	Store word as 2nd half of atomic swap
	load upper immed.	lui \$s1,20	$\$s1 = 20 * 2^{16}$	Loads constant in upper 16 bits
	and	and \$s1,\$s2,\$s3	$\$s1 = \$s2 \& \$s3$	Three reg. operands; bit-by-bit AND
	or	or \$s1,\$s2,\$s3	$\$s1 = \$s2 \$s3$	Three reg. operands; bit-by-bit OR
	nor	nor \$s1,\$s2,\$s3	$\$s1 = \sim(\$s2 \$s3)$	Three reg. operands; bit-by-bit NOR
	and immediate	andi \$s1,\$s2,20	$\$s1 = \$s2 \& 20$	Bit-by-bit AND reg with constant
	or immediate	ori \$s1,\$s2,20	$\$s1 = \$s2 20$	Bit-by-bit OR reg with constant
Conditional branch	shift left logical	sll \$s1,\$s2,10	$\$s1 = \$s2 \ll 10$	Shift left by constant
	shift right logical	srl \$s1,\$s2,10	$\$s1 = \$s2 \gg 10$	Shift right by constant
	branch on equal	beq \$s1,\$s2,25	if ($\$s1 == \$s2$) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne \$s1,\$s2,25	if ($\$s1 \neq \$s2$) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt \$s1,\$s2,\$s3	if ($\$s2 < \$s3$) $\$s1 = 1$; else $\$s1 = 0$	Compare less than; for beq, bne
	set on less than unsigned	sltu \$s1,\$s2,\$s3	if ($\$s2 < \$s3$) $\$s1 = 1$; else $\$s1 = 0$	Compare less than unsigned
	set less than immediate	slti \$s1,\$s2,20	if ($\$s2 < 20$) $\$s1 = 1$; else $\$s1 = 0$	Compare less than constant
Unconditional jump	set less than immediate unsigned	sltiu \$s1,\$s2,20	if ($\$s2 < 20$) $\$s1 = 1$; else $\$s1 = 0$	Compare less than constant unsigned
	jump	j 2500	go to 10000	Jump to target address
	jump register	jr \$ra	go to \$ra	For switch, procedure return
	jump and link	jal 2500	$\$ra = \text{PC} + 4$; go to 10000	For procedure call

FIGURE 2.1 MIPS assembly language revealed in this chapter. This information is also found in Column 1 of the MIPS Reference Data Card at the front of this book.

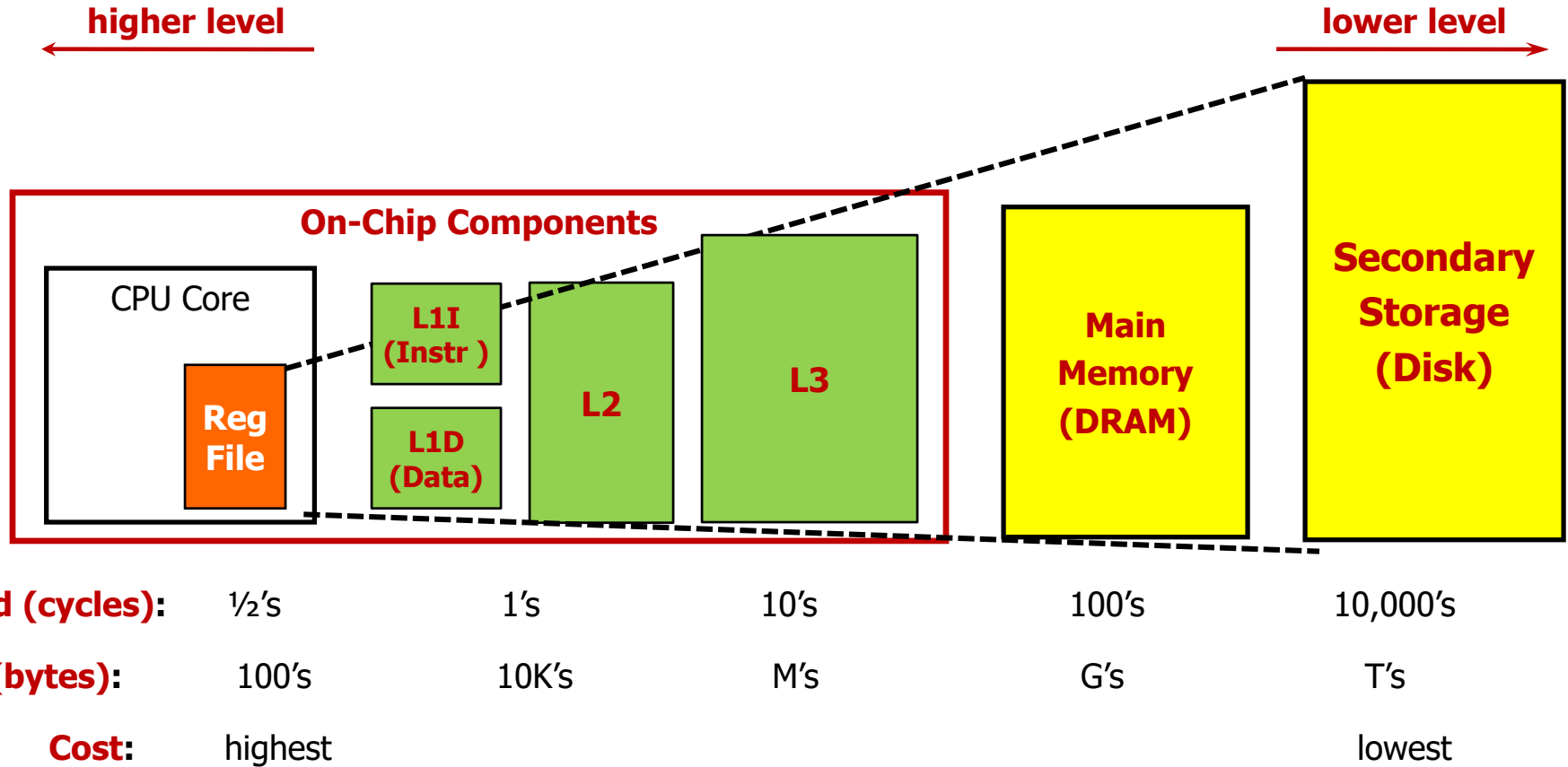
A Memory Hierarchy



**2nd Gen. Core i7
(2011)**



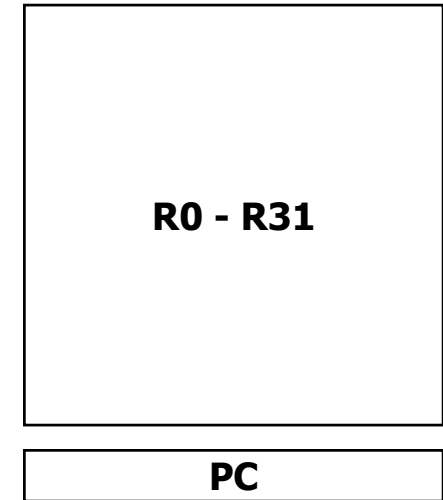
A Memory Hierarchy



MIPS Instruction Formats

- Instruction categories
 - Arithmetic and Logical (Integer)
 - Load/Store
 - Jump and Branch
 - Floating Point

Register file



- 3 Instruction formats: all 32 bits wide

opcode	rs	rt	rd	sa	funct
--------	----	----	----	----	-------

R format

opcode	rs	rt	immediate
--------	----	----	-----------

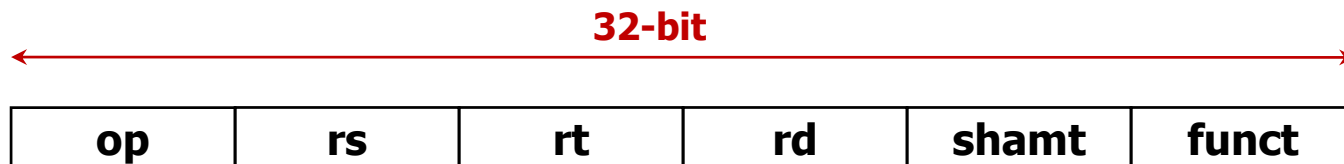
I format

opcode	jump target
--------	-------------

J format

MIPS Instruction Fields

- MIPS fields are given names to make them easier to refer to



op	6-bits	opcode that specifies the operation
rs	5-bits	register of the first source operand
rt	5-bits	register of the second source operand
rd	5-bits	register of the result's destination
shamt	5-bits	shift amount (for shift instructions)
funct	6-bits	function code augmenting the opcode

Overview of MIPS Operation

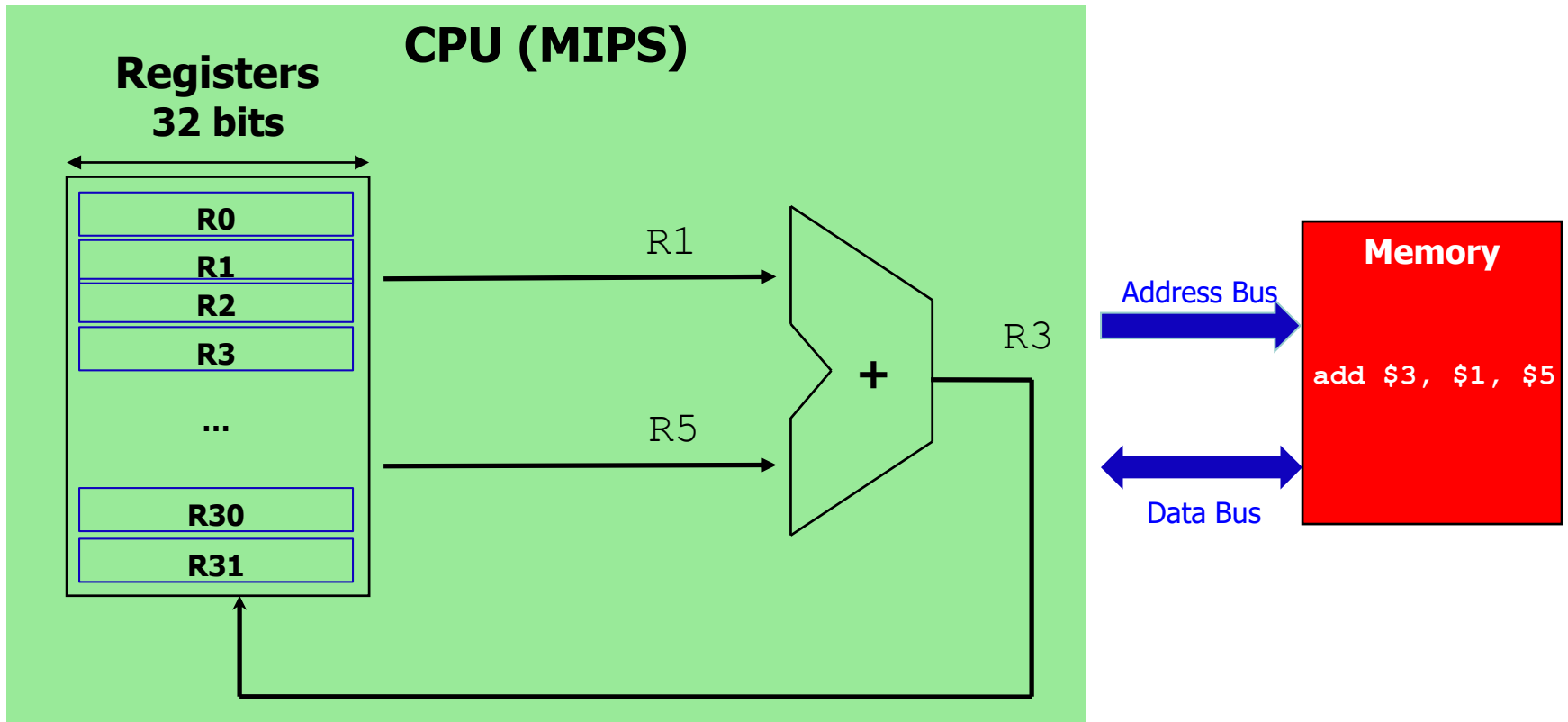
- MIPS arithmetic in assembly form

```
add    $3, $1, $5    # R3 = R1 + R5
```

- R1 and R5 are source operands, and R3 is destination
 - # indicate a comment, so assembler ignores it
- Operands of arithmetic instructions come from special locations called **registers** inside CPU or from the **immediate** field in instructions
 - All CPUs (x86, PowerPC, MIPS, ARM...) have registers inside
- **Registers are visible to the programmers**
 - MIPS has a **register file** consisting of 32 32-bit registers

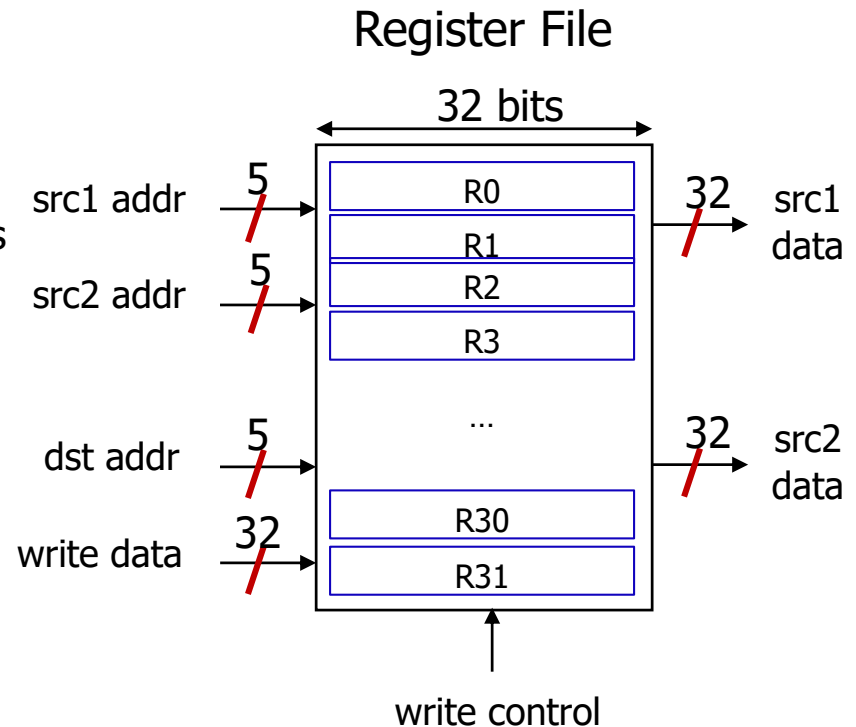
Simplified Version of CPU Internal

add \$3, \$1, \$5 # R3 = R1 + R5



MIPS Register File

- Registers are implemented with **flip-flops**
 - For example, one 32-bit register requires 32 flop-flops
- A set of architectural (programmer-visible) registers inside CPU is called **register file**
 - Register file can be implemented with flip-flops or SRAM
 - MIPS register file has 32 32-bit registers
 - Two read ports
 - One write port
 - **Register file access is much faster than main memory or cache** because there are a very limited number of registers and they reside inside CPU
 - So, compilers strive to use the register file when translating high-level code to assembly code



MIPS Register Convention

Name	Register Number	Usage	Preserve on call?
\$zero	0	constant 0 (hardwired)	n.a.
\$at	1	reserved for assembler	n.a.
\$v0 - \$v1	2-3	returned values	no
\$a0 - \$a3	4-7	arguments	yes
\$t0 - \$t7	8-15	temporaries	no
\$s0 - \$s7	16-23	saved values	yes
\$t8 - \$t9	24-25	temporaries	no
\$k0 - \$k1	26-27	Operating system (OS) temporaries	
\$gp	28	global pointer	yes
\$sp	29	stack pointer	yes
\$fp	30	frame pointer	yes
\$ra	31	return address	yes

Register File in Verilog

```
module regfile(input    clk,
               input    we,
               input  [4:0] ra1, ra2, wa,
               input  [31:0] wd,
               output [31:0] rd1, rd2);
```

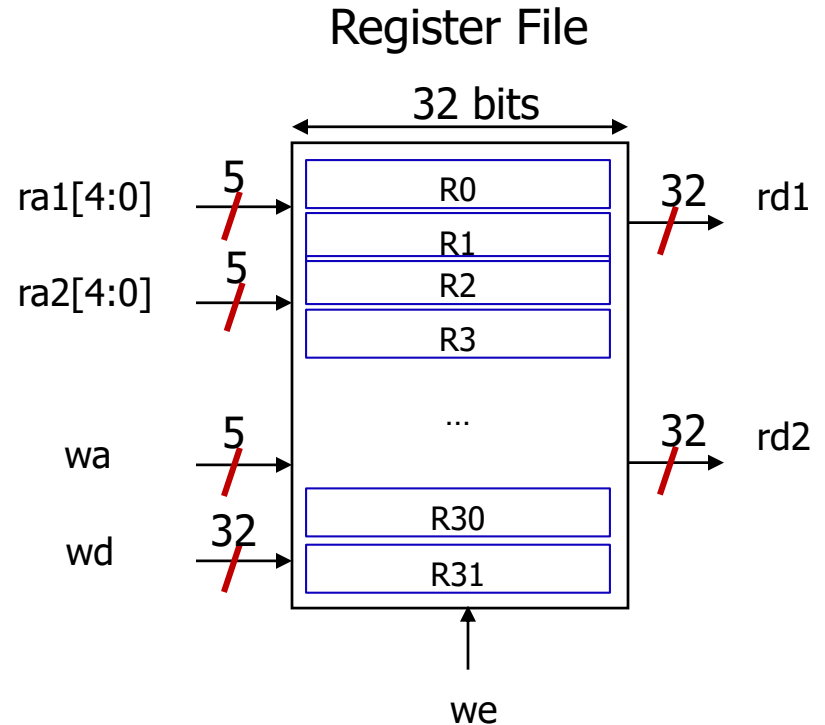
```
    reg [31:0] rf[31:0];
```

```
    // three ported register file
    // read two ports combinationaly
    // write third port on rising edge of clock
    // register 0 hardwired to 0
```

```
    always @(posedge clk)
        if (we) rf[wa] <= wd;
```

```
    assign rd1 = (ra1 != 0) ? rf[ra1] : 0;
    assign rd2 = (ra2 != 0) ? rf[ra2] : 0;
```

```
endmodule
```



Flip-Flop Version of Register File

```
module regfile ( input      clk,  
                 input      we,  
                 input  [4:0] ra1, ra2, wa,  
                 input  [31:0] wd,  
                 output reg [31:0] rd1, rd2);
```

```
    reg [31:0] R1;  
    reg [31:0] R2;  
    reg [31:0] R3;  
    reg [31:0] R4;  
    reg [31:0] R5;  
    reg [31:0] R6;  
    reg [31:0] R7;  
    reg [31:0] R8;  
    reg [31:0] R9;  
    reg [31:0] R10;  
    reg [31:0] R11;  
    reg [31:0] R12;  
    reg [31:0] R13;  
    reg [31:0] R14;  
    reg [31:0] R15;  
    reg [31:0] R16;  
    reg [31:0] R17;  
    reg [31:0] R18;  
    reg [31:0] R19;  
    reg [31:0] R20;  
    reg [31:0] R21;  
    reg [31:0] R22;  
    reg [31:0] R23;  
    reg [31:0] R24;  
    reg [31:0] R25;  
    reg [31:0] R26;  
    reg [31:0] R27;  
    reg [31:0] R28;  
    reg [31:0] R29;  
    reg [31:0] R30;  
    reg [31:0] R31;
```

```
    always @(*)  
    begin  
        case (ra1[4:0])  
            5'd0: rd1 = 32'b0;  
            5'd1: rd1 = R1;  
            5'd2: rd1 = R2;  
            5'd3: rd1 = R3;  
            5'd4: rd1 = R4;  
            5'd5: rd1 = R5;  
            5'd6: rd1 = R6;  
            5'd7: rd1 = R7;  
            5'd8: rd1 = R8;  
            5'd9: rd1 = R9;  
            5'd10: rd1 = R10;  
            5'd11: rd1 = R11;  
            5'd12: rd1 = R12;  
            5'd13: rd1 = R13;  
            5'd14: rd1 = R14;  
            5'd15: rd1 = R15;  
            5'd16: rd1 = R16;  
            5'd17: rd1 = R17;  
            5'd18: rd1 = R18;  
            5'd19: rd1 = R19;  
            5'd20: rd1 = R20;  
            5'd21: rd1 = R21;  
            5'd22: rd1 = R22;  
            5'd23: rd1 = R23;  
            5'd24: rd1 = R24;  
            5'd25: rd1 = R25;  
            5'd26: rd1 = R26;  
            5'd27: rd1 = R27;  
            5'd28: rd1 = R28;  
            5'd29: rd1 = R29;  
            5'd30: rd1 = R30;  
            5'd31: rd1 = R31;  
        endcase  
    end
```

Flip-Flop Version of Register File

```

always @(*)
begin
    case (ra2[4:0])
        5'd0: rd2 = 32'b0;
        5'd1: rd2 = R1;
        5'd2: rd2 = R2;
        5'd3: rd2 = R3;
        5'd4: rd2 = R4;
        5'd5: rd2 = R5;
        5'd6: rd2 = R6;
        5'd7: rd2 = R7;
        5'd8: rd2 = R8;
        5'd9: rd2 = R9;
        5'd10: rd2 = R10;
        5'd11: rd2 = R11;
        5'd12: rd2 = R12;
        5'd13: rd2 = R13;
        5'd14: rd2 = R14;
        5'd15: rd2 = R15;
        5'd16: rd2 = R16;
        5'd17: rd2 = R17;
        5'd18: rd2 = R18;
        5'd19: rd2 = R19;
        5'd20: rd2 = R20;
        5'd21: rd2 = R21;
        5'd22: rd2 = R22;
        5'd23: rd2 = R23;
        5'd24: rd2 = R24;
        5'd25: rd2 = R25;
        5'd26: rd2 = R26;
        5'd27: rd2 = R27;
        5'd28: rd2 = R28;
        5'd29: rd2 = R29;
        5'd30: rd2 = R30;
        5'd31: rd2 = R31;
    endcase
end

```

```

always @(posedge clk)
begin
    if (we)
    begin
        case (wa[4:0])
            5'd0: ;
            5'd1: R1 <= wd;
            5'd2: R2 <= wd;
            5'd3: R3 <= wd;
            5'd4: R4 <= wd;
            5'd5: R5 <= wd;
            5'd6: R6 <= wd;
            5'd7: R7 <= wd;
            5'd8: R8 <= wd;
            5'd9: R9 <= wd;
            5'd10: R10 <= wd;
            5'd11: R11 <= wd;
            5'd12: R12 <= wd;
            5'd13: R13 <= wd;
            5'd14: R14 <= wd;
            5'd15: R15 <= wd;
            5'd16: R16 <= wd;
            5'd17: R17 <= wd;
            5'd18: R18 <= wd;
            5'd19: R19 <= wd;
            5'd20: R20 <= wd;
            5'd21: R21 <= wd;
            5'd22: R22 <= wd;
            5'd23: R23 <= wd;
            5'd24: R24 <= wd;
            5'd25: R25 <= wd;
            5'd26: R26 <= wd;
            5'd27: R27 <= wd;
            5'd28: R28 <= wd;
            5'd29: R29 <= wd;
            5'd30: R30 <= wd;
            5'd31: R31 <= wd;
        endcase
    end
end

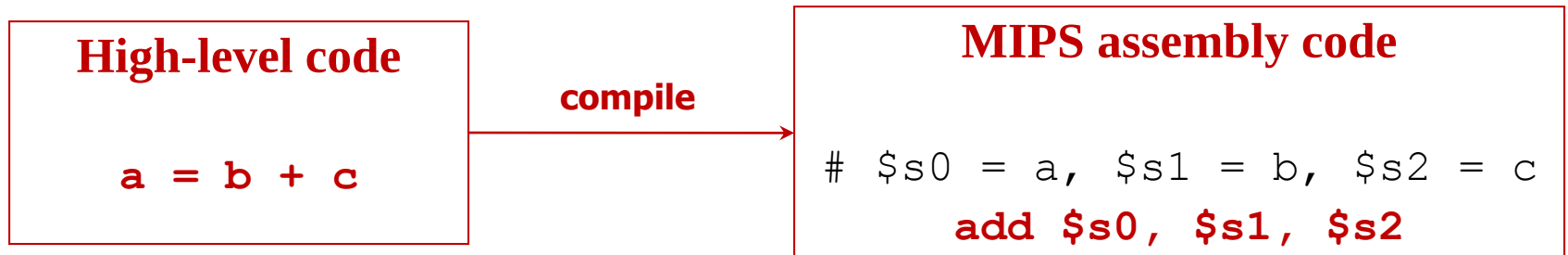
```

MIPS Instructions

- For the complete instruction set, refer to the “Appendix B.10 MIPS R2000 Assembly Language”
 - For detailed information on the MIPS instruction set, refer to the Appendix A (page 469) in MIPS R4000 specification linked on the class page
- We are going to cover essential and important instructions in this course
 - Again, if you completely understand one CPU, it is pretty straightforward to understand other CPUs
 - For the term project, you should implement hardware (that is, CPU) that is able to execute those essential MIPS instructions
 - Let’s go over MIPS instructions one by one

MIPS Arithmetic Instructions

- MIPS Arithmetic instructions include `add`, `sub`, `addi`, `addiu`, `mult`, `div` and some more
 - Check out the appendix for the list of all arithmetic instructions



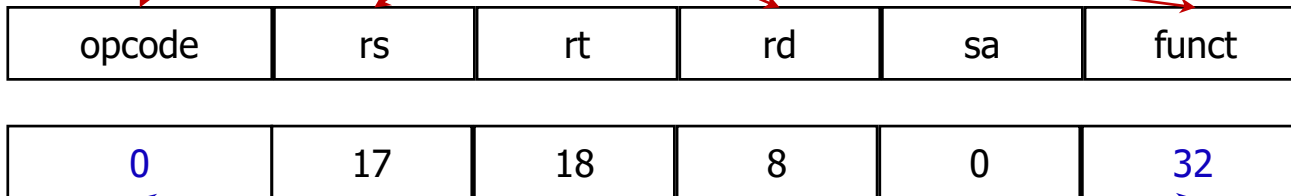
add

- R format instruction

add rd, rs, rt

- Example:

add \$t0, \$s1, \$s2 # \$t0 <= \$s1 + \$s2



MIPS architect defines the opcode and function

binary 000000 10001 10010 01000 00000 100000
 000000 10001 10010 01000 00000 100000

hexadecimal 0x0232 4020

Name	Register Number
\$zero	0
\$at	1
\$v0 - \$v1	2-3
\$a0 - \$a3	4-7
\$t0 - \$t7	8-15
\$s0 - \$s7	16-23
\$t8 - \$t9	24-25
\$gp	28
\$sp	29
\$fp	30
\$ra	31

sub

- R format instruction

sub rd, rs, rt

- Example:

sub \$t2, \$s3, \$s4 # \$t2 <= \$s3 - \$s4

opcode	rs	rt	rd	sa	funct
0	19	20	10	0	34

MIPS architect defines the opcode and function

binary 000000 10011 10100 01010 00000 100010
 000000 10011 10100 01010 00000 100010

hexadecimal 0x0274 5022

Name	Register Number
\$zero	0
\$at	1
\$v0 - \$v1	2-3
\$a0 - \$a3	4-7
\$t0 - \$t7	8-15
\$s0 - \$s7	16-23
\$t8 - \$t9	24-25
\$gp	28
\$sp	29
\$fp	30
\$ra	31

Immediate

- R-format instructions have all 3 operands in registers
- In I-format instructions, one operand can be stored in instruction itself
 - They are called **immediates** because they are immediately available from the instructions
 - They do not require a register or memory access
 - 16-bit immediate field in MIPS instructions limits values to the range of $(-2^{15} \sim +2^{15}-1)$ since it uses 2's complement



I format

Revisiting 2's Complement Number

- In hardware design of computer arithmetic, the 2s complement number provides a convenient and simple way to do addition and subtraction of **unsigned and signed numbers**
- Given an n-bit number N in binary, the 2s complement of N is defined as
$$\begin{aligned} &2^n - N \text{ for } N \neq 0 \\ &0 \quad \text{for } N = 0 \end{aligned}$$
 - Example:
 - With a 4-bit, 3 is 4'b0011 and 2's complement of 3: $2^4 - 3 = 4'b1101$
- A fast way to get a 2s complement number is to **flip all the bits and add 1**

Number System Comparison with N-bit

Number System	Range
Unsigned	$[0, 2^N-1]$
Sign/Magnitude	$[-(2^{N-1}-1), 2^{N-1}-1]$
2's Complement	$[-2^{N-1}, 2^{N-1}-1]$

- Thus, 16-bit can represent a range of
 - Unsigned: $[0 \sim +(2^{16}-1)] = [0 \sim +65535]$
 - Sign/Magnitude: $[-(2^{16-1}-1) \sim +(2^{16-1}-1)] = [-32767 \sim +32767]$
 - 2's complement: $[-2^{16-1} \sim +2^{16-1}-1] = [-32768 \sim +32767]$

addi

- I format instruction

addi rt, rs, imm

- Example:

addi \$t0, \$s3, -12 # \$t0 = \$s3 + (-12)

opcode	rs	rt	immediate
8	19	8	-12

binary 001000 10011 01000 11111 11111 110100

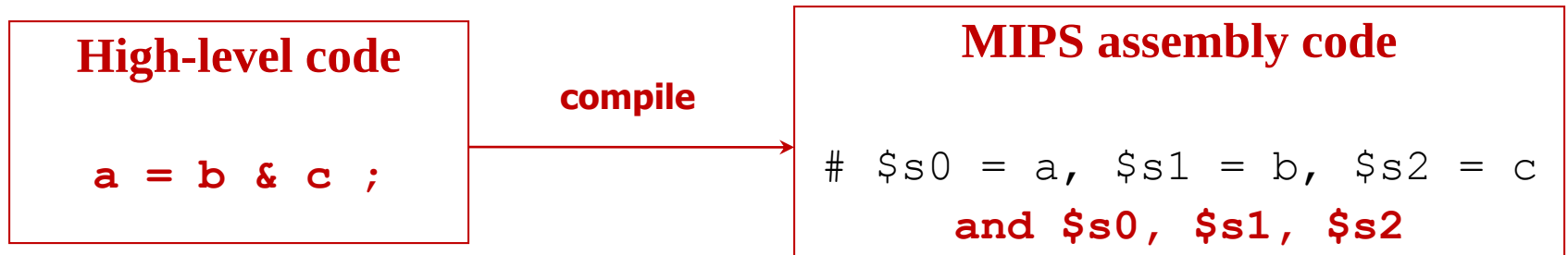
001000 10011 01000 11111 11111 110100

hexadecimal 0x2268 FFF4

Name	Register Number
\$zero	0
\$at	1
\$v0 - \$v1	2-3
\$a0 - \$a3	4-7
\$t0 - \$t7	8-15
\$s0 - \$s7	16-23
\$t8 - \$t9	24-25
\$gp	28
\$sp	29
\$fp	30
\$ra	31

MIPS Logical Instructions

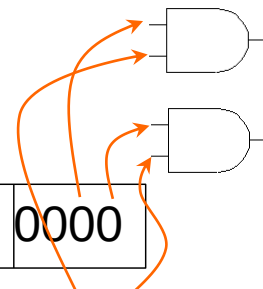
- MIPS logical instructions include `and`, `andi`, `or`, `ori`, `xor`, `nor` etc
- Logical instructions operate bit-by-bit on 2 source operands and write the result to the destination register



Logical Instruction Examples

Source Registers

\$s1	1111	1111	1111	1111	0000	0000	0000	0000
\$s2	0100	0110	1010	0001	1111	0000	1011	0111



Assembly Code

```
and $s3, $s1, $s2  
or  $s4, $s1, $s2  
xor $s5, $s1, $s2  
nor $s6, $s1, $s2
```

Result

\$s3								
\$s4								
\$s5								
\$s6								

Logical Instruction Examples

Source Registers

\$s1	1111	1111	1111	1111	0000	0000	0000	0000
\$s2	0100	0110	1010	0001	1111	0000	1011	0111

Assembly Code

```
and $s3, $s1, $s2  
or  $s4, $s1, $s2  
xor $s5, $s1, $s2  
nor $s6, $s1, $s2
```

Result

\$s3	0100	0110	1010	0001	0000	0000	0000	0000
\$s4	1111	1111	1111	1111	1111	0000	1011	0111
\$s5	1011	1001	0101	1110	1111	0000	1011	0111
\$s6	0000	0000	0000	0000	0000	1111	0100	1000

AND, OR, and NOR Usages

- `and`, `or`, `nor`
 - `and` is useful for masking bits
 - Example: mask all but the least significant byte of a value:
 $0xF234012F \text{ AND } 0x000000FF = 0x0000002F$
 - `or` is useful for combining bit fields
 - Example: combine $0xF2340000$ with $0x000012BC$:
 $0xF2340000 \text{ OR } 0x000012BC = 0xF23412BC$
 - `nor` is useful for inverting bits:
 - Example: $A \text{ NOR } \$0 = \text{NOT } A$

and, or, nor

- R format instruction

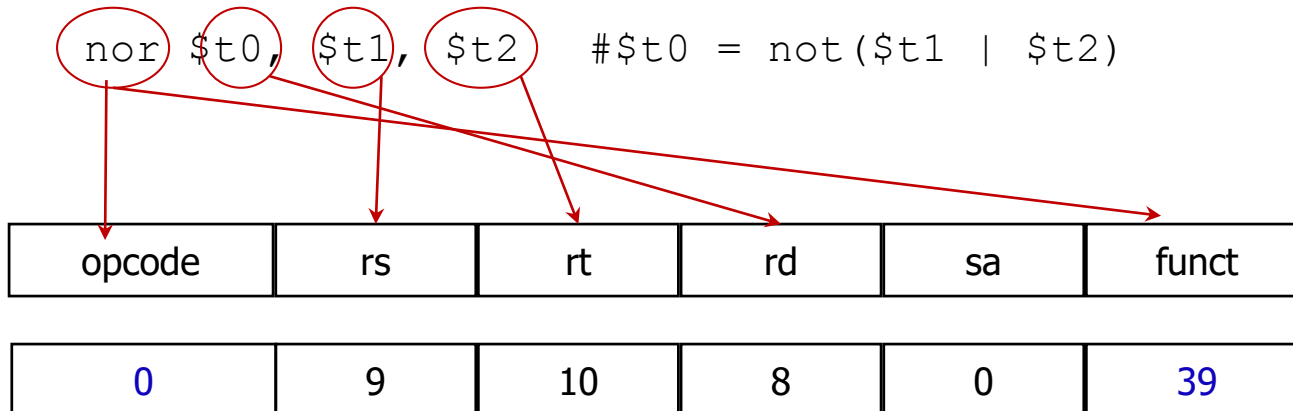
and (or, nor) rd, rs, rt

- Examples:

and \$t0, \$t1, \$t2 #\$t0 = \$t1 & \$t2

or \$t0, \$t1, \$t2 #\$t0 = \$t1 | \$t2

nor \$t0, \$t1, \$t2 #\$t0 = not(\$t1 | \$t2)



binary 000000 01001 01010 01000 00000 100111

000000 01001 01010 01000 00000 100111

hexadecimal 0x012A 4027

Name	Register Number
\$zero	0
\$at	1
\$v0 - \$v1	2-3
\$a0 - \$a3	4-7
\$t0 - \$t7	8-15
\$s0 - \$s7	16-23
\$t8 - \$t9	24-25
\$gp	28
\$sp	29
\$fp	30
\$ra	31

andi, ori

- I format instruction

andi(ori) rt, rs, imm

- Example:

andi \$t0, \$t1, 0xFF00 # \$t0 = \$t1 & ff00

ori \$t0, \$t1, 0xFF00 # \$t0 = \$t1 | ff00

opcode	rs	rt	immediate
13	9	8	0xFF00

binary 001101 01001 01000 11111 11100 000000

001101 01001 01000 11111 11100 000000

hexadecimal 0x3528 FF00

Name	Register Number
\$zero	0
\$at	1
\$v0 - \$v1	2-3
\$a0 - \$a3	4-7
\$t0 - \$t7	8-15
\$s0 - \$s7	16-23
\$t8 - \$t9	24-25
\$gp	28
\$sp	29
\$fp	30
\$ra	31

Sign Extension & Zero Extension

- Most MIPS instructions **sign-extend** the immediate
 - For example, `addi` does sign-extension to support both positive and negative immediates
- An exception to the rule is that logical operations (`andi`, `ori`, `xori`) place 0's in the upper half
 - This is called **zero extension**

Revisiting Basic Shifting

- Shift types
 - Logical (or unsigned) shift
 - Arithmetic (or signed) shift
- Shift directions
 - Left (multiply by powers of 2)
 - Right (divide by powers of 2)
 - Take floor value if the result is not an integer
 - Floor value of X (or $\lfloor X \rfloor$) is the largest integer less than or equal to X
 - $\lfloor 5/2 \rfloor = 2$
 - $\lfloor -3/2 \rfloor = -2$

Revisiting Logical Shift

- Logical shift left
 - MSB: shifted out
 - LSB: shifted in with a 0
 - Examples:
 - $(11001011 \ll 1) = 10010110$
 - $(11001011 \ll 3) = 01011000$
- Logical shift right
 - MSB: shifted in with a 0
 - LSB: shifted out
 - Examples:
 - $(11001011 \gg 1) = 01100101$
 - $(11001011 \gg 3) = 00011001$
- Logical shifts are useful to perform multiplication or division of **unsigned** integer by powers of two
 - Logical shift right takes floor value if the result is not integer

Revisiting Arithmetic Shift

- Arithmetic shift left
 - MSB: shifted out, however, be aware of overflow/underflow
 - LSB: shifted in with a 0
 - Examples:
 - $(1100 \ll 1) = 1000$
 - $(1100 \ll 3) = 0000$ (Incorrect!) $\Rightarrow$ Underflow
- Arithmetic shift right
 - MSB: Retain its sign bit
 - LSB: Shifted out
 - Examples:
 - $(1100 \gg 1) = 1110$ (Retain sign bit)
 - $(1100 \gg 3) = 1111$ ($\lfloor -4/8 \rfloor = -1$) $\Rightarrow$ Floor value of -0.5
- Arithmetic shifts can be useful as efficient ways of performing multiplication or division of **signed** integers by powers of two
 - Arithmetic shift right takes floor value if the result is not integer

MIPS Shift Instructions

- MIPS shift instructions include `sll`, `srl`, `sra`, `sllv`, `srlv` and `srav`
- Shift-left operation multiplies a number by powers of 2
- Shift-right operation divides a number by powers of 2

High-level code

```
int a, b, c;  
b = a * 4  
c = a / 4
```

compile

MIPS assembly code

```
# $s0 = a, $s1 = b, $s2 = c  
sll $s1, $s0, 2  
sra $s2, $s0, 2
```

sll, srl, sra

- R format instructions
- Shift instructions shift the value in a register left or right by up to 31 bits (5-bit shamt field)

sll rd, rt, shamt: shift left logical

srl rd, rt, shamt: shift right logical

sra rd, rt, shamt: shift right arithmetic (sign-extension)

- Examples:

sll \$t0, \$s1, 4 # \$t0 = \$s1 << 4 bits

srl \$t2, \$s0, 8 # \$t2 = \$s0 >> 8 bits

sra \$s3, \$s1, 4 # \$s3 = \$s1 >>> 4 bits

opcode	rs	rt	rd	sa	funct
0	0	17	19	4	3

Name	Register Number
\$zero	0
\$at	1
\$v0 - \$v1	2-3
\$a0 - \$a3	4-7
\$t0 - \$t7	8-15
\$s0 - \$s7	16-23
\$t8 - \$t9	24-25
\$gp	28
\$sp	29
\$fp	30
\$ra	31

Binary ?
Hexadecimal: ?

sllv, srlv, srav

- R format instructions

- MIPS also has **variable-shift** instructions

sllv rd, rt, rs: shift left logical variable

srlv rd, rt, rs: shift right logical variable

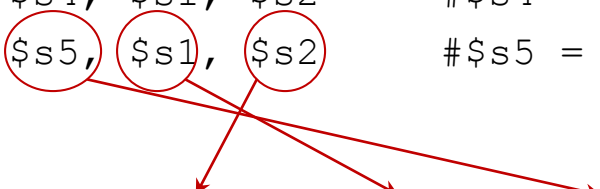
srav rd, rt, rs: shift right arithmetic variable

- Examples:

sllv \$s3, \$s1, \$s2 # \$s3 = \$s1 << \$s2

srlv \$s4, \$s1, \$s2 # \$s4 = \$s1 >> \$s2

srav \$s5, \$s1, \$s2 # \$s5 = \$s1 >>> \$s2



opcode	rs	rt	rd	sa	funct
0	18	17	21	0	7

Name	Register Number
\$zero	0
\$at	1
\$v0 - \$v1	2-3
\$a0 - \$a3	4-7
\$t0 - \$t7	8-15
\$s0 - \$s7	16-23
\$t8 - \$t9	24-25
\$gp	28
\$sp	29
\$fp	30
\$ra	31

Binary ?

Hexadecimal: ?